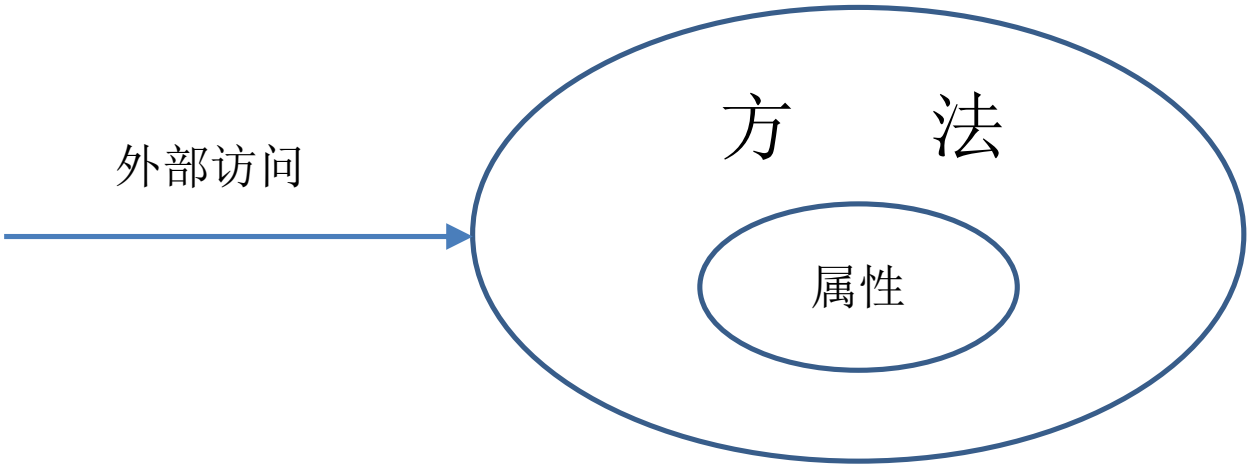
Chapter 5 Hiding the Implementation

本章内容

- C++中的封装
- Java中的封装
- 语法和语义封装

为什么需要信息隐藏

- 外部程序是否能够直接访问对象的内部属性数据？
- 是否应当加以限制？
- 好处、坏处？



为什么需要信息隐藏

- 例子
  - 人的年龄、性别等属性数据特性：初始化一次、只读

```
class Student
{
  private  Date  birthday;      //定义出生年月为私有成员,外界不可访问。
  Student(Date birthday)        //构造方法,对出生年月进行初始化赋值一次
  {
      this.birthday=birthday;
  }
  public  Date  getBirthday()  //公共方法,对出生年月进行读取
  {
      return birthday;
  }
}
```

为什么需要信息隐藏

- 封装(encapsulation)，将对象的状态信息隐藏在对应内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问
- 是否把事情搞复杂了？

为什么需要信息隐藏

- 把暴露的数据封装起来，尽可能的让对象管理它们自己的状态，因为过多的依存性会造就紧耦合(highly coupled)系统，使得任何一点小小改动都可能造成许多无法预料的结果。
- 需要控制对内部成员访问的两个理由：
  - 让客户远离一些他们不需要使用的数据和方法，它们只对抽象数据类型的内部处理来说是必须的。对客户程序员提供方便。
  - 允许库的设计者改变抽象数据类型内部的实现，而不必担心对客户程序产生影响。

为什么需要信息隐藏

- 类的公用接口与私有实现的分离
  - 公用接口：公用成员函数
  - 私有实现：类中被操作的数据是私有的，类的功能的实现细节对用户是隐蔽的
- 好处
  - **隐藏细节**：使用者只需要了解如何通过类的接口使用类，而不用关心类的内部数据结构和数据组织方法；
  - **便于修改**：如果想扩充或修改类的功能，只需修改该类中有关的数据成员及相关的成员函数，程序中类以外的部分可以不必修改；提高了代码的可维护性；
  - **便于维护**：只要类的接口没有改变，对内部实现的修改不会引起程序的其他部分的修改；
  - **便于调试**：检查调试方便，数据读写错误时，只需检查相关的成员函数

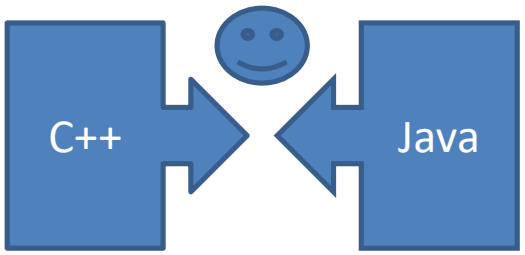
Java类的访问控制

- 一个类的成员可以有以下的修饰符：
  - private
  - protected
  - public
  - default

访问控制级别由小到大

C++类的访问控制

- 一个类的成员可以有以下的修饰符：
  - private
    - 只能被类内部的成员访问
  - protected
    - 只能被类内部和其子类访问
  - public
    - 能被所有类访问



Access Levels(Java & C++)

Java Class

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no identifier	Y	Y	N	N
private	Y	N	N	N

C++

Modifier	Class	Package	Subclass	World
public	Y	N/A	Y	Y
protected	Y	N/A	Y	N
no identifier	Y		N	N
private	Y	N/A	N	N

C++类的访问控制

- 公有派生

基类成员属性	派生类中	派生类外
public	可以引用	可以引用
protected	可以引用	不可引用
private	不可引用	不可引用

C++类的访问控制

- 私有派生

基类成员属性	派生类	派生类外
public	可以引用	不可引用
protected	可以引用	不可引用
private	不可引用	不可引用

C++类的访问控制

- 保护派生

基类成员属性	派生类	派生类外
public	可以引用	不可引用
protected	可以引用	不可引用
private	不可引用	不可引用

C++类的访问控制

- 完善Stash
  - 对比C05:Stash.h和C04:CppLib.h
- 完善Stack
  - 对比C05:Stack2.h和C04:Stack.h

嵌套类

- 定义嵌套类目的在于隐藏类名，减少全局的标识符，从而限制用户能否使用该类建立对象。这样可以提高类的抽象能力，并且强调了两个类(外围类和嵌套类)之间的主从关系
- 外围类对嵌套类的成员无特殊访问权，反之亦然
- 嵌套类声明的位置决定了作用域和可见性及访问控制规则

声明位置	包含它的类是否可以使用它	从包含它的类派生而来的类是否可以使用它	在外部是否可以使用
public	可以引用	可以引用	引用要加类限定符
protected	可以引用	可以引用	不可引用
private	可以引用	不可引用	不可引用

局部类

- 局部类：在函数内部定义的类；
- 局部类只在定义它的局部域内可见，在定义该类的局部域外没有语法能够引用局部类的成员，所有成员函数必须定义在类体内；
- 没有语法能够在名字空间域内定义局部类的成员，所以也不允许局部类声明静态数据成员
- 局部类只能看到外围函数内定义的类型名，静态变量和枚举，外围函数的局部变量是看不到的；

友元

- 如果程序员想显式地允许不属于当前结构的一个成员访问当前结构中的数据，可以在该结构内部声明设个函数为友元(friend)
- Who can access my private implementation?

- Friend.cpp

友元

- 友元函数是一种定义在类外部的普通函数，其特点是能够访问类中私有成员和保护成员，即类的访问权限的限制对其不起作用。
  - 友元函数需要在类体内进行说明，在前面加上关键字friend。
  - 一般格式为：friend <type> FuncName(<args>);

友元

- 友元函数不是类的成员函数，用法也与普通的函数完全一致，只不过它能访问类中所有的数据。

- 一个类的友元可以自由地用该类中的所有成员（公有的、私有的和保护的），而一般函数只能访问类中的公有成员。

```
class A{
        float  x,y;
public:
A(float a, float b){  x=a; y=b;}
float  Sum(){  return  x+y;  }
friend  float Sum(A &a){     return   a.x+a.y; }
};
int main(void)
{  A   t1(4,5),t2(10,20);
    cout<<t1.Sum()<<endl;
    cout<<Sum(t2)<<endl;
}
```

友元函数只能用对象名引用类中的数据。

成员函数

友元函数

私有数据

成员函数的调用，利用对象名调用

友元函数的调用，直接调用

友元

- 友元函数近似于普通的函数，它不带有this指针，因此必须将对象名或对象的引用作为友元函数的参数，这样才能访问到对象的成员。

- 友元函数不受类中访问权限关键字的限制，可以把它放在类的私有部分，公有部分或保护部分，其作用都是一样的。换言之，在类中对友元函数指定访问权限是不起作用的。

- 友元函数的作用域与一般函数的作用域相同。

友元

- 友元函数破坏了类的封装性和隐蔽性，使得非成员函数可以访问类的私有成员。
- 谨慎使用友元函数
- 通常使用友元函数来取对象中的数据成员值，而不修改对象中的成员值，则肯定是安全的。

大多数情况是友元函数是某个类的成员函数，即A类中的某个成员函数是B类中的友元函数，这个成员函数可以直接访问B类中的私有数据。这就实现了类与类之间的沟通。

```
class A{
...
void fun( B &);
};
```

既是类A的成员函数

```
class B{
...
friend void fun( B &);
};
```

又是类B的友元函数

注意：一个类的成员函数作为另一个类的友元函数时，应先定义友元函数所在的类。

友元

```
class A{
    .....
    friend class B;
}
```

类B是类A的友元

```
class B{
    .....
}
```

类B可以自由使用类A中的成员

对于类B而言，类A是透明的

类B必须通过类A的对象使用类A的成员

友元

- 不管是按哪一种方式派生，基类的私有成员在派生类中都是不可见的。
- 如果在一个派生类中要访问基类中的私有成员，可以将这个派生类声明为基类的友元。

```
class Base {
    friend class  Derive;
     .....
}
```

```
class Derive {
     .....
}
```

直接使用Base中的私有成员

嵌套友元

- 嵌套结构并不能自动获得访问private成员的权限。要获得父结构私有成员的访问权限，必须遵循特定的规则：
  - 1-先声明一个嵌套的结构；
  - 2-声明friend;
  - 3-定义友元结构

- NestFriend.cpp

友元

- 注意事项：
  - 友元关系是单向的
  - 友元关系不能被传递
  - 友元关系不能被继承

为什么使用友元

- 目的：
  - 允许外面的类或函数去访问类的私有变量和保护变量，从而使两个类共享同一函数；
  - 在实现类之间数据共享时，减少系统开销，提高效率。
- 优点：能够提高效率，表达简单、清晰；
- 缺点：友元函数破坏了封装机制；不是面向对象

句柄类（handle class）

- 进一步的隐藏
  - 对于private实现部分的隐藏；
  - 开发初期非常有意义：只要接口不变，头文件就不需要改动；
  - C05:Handle.h

Java中的内部类

Inner class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.

- An inner class can reference the data and methods defined in the outer class in which it nests, even the ones modified by private. so you do not need to pass the reference of the outer class to the constructor of the inner class.
- Frequently used with GUI handling

```java
//外部类
class Out {
    private int age = 12;

    //内部类
    class In {
        public void print() {
            System.out.println(age);
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        Out.In in = new Out().new In();
        in.print();
        //或者采用下种方式访问
        /*
        Out out = new Out();
        Out.In in = out.new In();
        in.print();
        */
    }
```

Java中的内部类

- 成员内部类（Member Inner Class）
  - 成员内部类可以访问外部类的静态与非静态的方法和成员变量
- 静态内部类
  - 类定义时加上static关键字，只能访问外部类的静态成员变量与静态方法
- 局部内部类
  - 定义在方法中，也只能在方法中使用。类似于局部变量，不能定义为public，protected，private或者static类型。只能访问方法中定义的final类型的局部变量。
- 匿名内部类
  - 隐式地继承了一个父类或者实现了一个接口。匿名内部类使用得比较多，通常是作为一个方法参数

Java中的内部类

- 内部类是个编译时的概念，一旦编译成功后，它就与外围类属于两个完全不同的类（当然他们之间还是有联系的）。对于一个名为OuterClass的外围类和一个名为InnerClass的内部类，在编译成功后，会出现这样两个class文件：OuterClass.class和OuterClass$InnerClass.class。

Java：成员类

- 成员类的显著特性就是成员类能访问它的外部类实例的任意字段与方法。方便一个类对外提供一个公共接口的实现是成员类的典型应用。

Java：成员类特点

- 能够访问到包含其类的所有数据成员
- 实例化时需要有一个包含类的示例
  - 由包含类实例化
    - 如上例
  - 包含类的实例：
    - out.new Inner();

Java:局部类

- 将一个类的定义和实例化移至方法体中，这个类成为局部类

Java:局部类

```
// This method creates and returns an Enumeration object
public java.util.Enumeration enumerate() {

  // Here's the definition of Enumerator as a local class
  class Enumerator implements java.util.Enumeration {
    Linkable current;
    public Enumerator() { current = head; }
    public boolean hasMoreElements() {  return (current != null); }
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  }

  // Now return an instance of the Enumerator class defined directly above
  return new Enumerator();
}
```

Java:局部类

- 局部类只在被定义的块中可见
- 局部类不能被声明为public, protected, private, 或 static
- 局部类不能够包含静态成员
- 不能定义为接口
- 局部类可以使用所定义的块中可见的，声明为final的局部变量。

良好的类接口

- 类的接口应该展现一致的抽象层次
  - 与当前接口抽象概念无关的函数不要放到接口中
- 尽可能地限制类成员的可访问性
  - 类的使用者知道得越少，出问题的的可能就越小
  - 语法封装public、protected、private、friend等
- 不要对类的使用者作出任何假设，除非在接口设计中有过明确说明
  - 从接口函数的参数上看，接口开发者不应该假定传入的参数合法性及合理性；从调用时序来看，不应该假定接口调用者在调用当前函数之前已经进行其他的依赖调用。
- 语义封装
  - 接口的使用者除了接口本身的释义之外，不应该去了解接口的实现细节，并假定这些细节一定会执行